Coding Guidelines

This page extends the general Go coding references mentioned in the Fabric Coding Guidelines documentation with suggestions specifically tailored to the Hyperledger Fabric codebase.

Content adapted from a presentation from Fabric maintainer Matt Sykes.

Disclaimers

- These are guidelines, not rules.
- Context matters; use judgement.
- Just because we can doesn't mean we should.
- Just because we should doesn't mean we must.

What we want

- Simplicity
- Readability
- Flexibility
- Consistency
- Maintainability

Avoid package level state

Package level state is shared global state. Any global state that can be accessed via an API becomes a latent dependency for all importing code.

Global state also impedes testing. If the state is easily mutated, tests that access the state cannot be executed in parallel and this can increase the test execution times.

When global state is protected by "initialize once" semantics, it's impossible for tests to execute construct and exercise different configurations.

Instead of package level state, convert the package variables to fields on structs. Instantiate instances of the structs as appropriate, and wire the structs to the code that depends on them.

Appropriate use of interfaces

Interfaces should generally be defined on the consumer side.

- Exceptions: It is acceptable to define interfaces with the provider (or at a common place) when more than one implementation will be provided or when a large number of consumers will need the same interface.

Avoid unnecessary interfaces to keep the code simpler and more readable

- Interfaces should be added for specific reasons, for example if code logic cannot be tested without a mock or fake.
- It is acceptable to test with real implementations if a large amount of the code logic can be driven with real implementations.
- Don't define an interface simply to achieve 100% test coverage.
- Generally it makes sense to add interfaces between larger modules rather than between every layer.

Prefer small interfaces

Smaller interfaces are generally more useful. Simple interfaces, especially those with one method, are far more likely to be reused across packages and satisfied by more types.

The canonical examples here are `io.Reader`, `io.Writer`, and the empty interface.

When large interfaces are defined, it is far less likely that the interfaces will be reused. When an interface is defined with the implementation, it's extremely unlikely that any other type will satisfy it.

Accept interfaces, return structs (concrete types)

An interface is a contract that describes behavior. Unlike Java, in Go, a type that implements all of the operations of an interface satisfies the interface.

The behavior of an implementation is defined by the operations it supports; it simply doesn't need an additional layer of indirection to reiterate that.
When implementing a function, we want to declare the required behavior of our dependencies. This is accomplished by defining the smallest interface that encapsulates that behavior. By decoupling the behavior from the implementation, users of our function can then provide any implementation that satisfies that contract.

This rule is especially true of constructors.

Use care to avoid interface pollution. If an appropriately scoped interface exists in a package you’re already consuming or the standard library, use it instead of defining yet another.

Avoid the factory pattern

The factory pattern is often used to defer the instantiation of a concrete type that satisfies some interface.

While there are plenty of scenarios where it’s appropriate to use factories and shared interfaces, the introduction should be deferred until necessary. We should never start out assuming that a factory is needed yet only support a single concrete implementation.

We should also try to limit the usage of a factory implementation to a single package.

Exported functions use exported types

When a function is exported, the arguments and return values should use exported types. If the caller can’t declare a var of the same type as a parameter or a returned value, it impacts their flexibility.

We often find constructors returning unexported concrete types and functions that accept arguments associated with an unexported interface. This should be avoided.

Explicitly manage dependencies

- Pass instances of dependencies to constructors and functions
- Pass configuration to constructors
- Use consumer side interfaces to define dependency behavior
- Do not get, create, or instantiate a dependency in a constructor
- Avoid the `_Support_ pattern that embeds all dependencies behind an aggregated interface.
- Avoid carrying a dependency’s dependencies

Be aware that time is a latent dependency. When time is relevant to the behavior of an object, use an explicit clock dependency so it can be controlled in test.

Exported fields are not evil

Consider dependencies. If we explicitly manage them, we need to provide them when creating an instance. We have two choices:

- a constructor function
- a struct literal

In both cases, the dependencies come from our callers. If there's no need for custom construction logic, just dependency wiring, enable the user to define a struct literal with the dependencies.

This does not mean that _all_ fields should be exported.

Because we want to hide implementation details, we often go out of our way to hide all fields of a struct. This isn't necessarily a bad thing, but it's not always the best thing.

Avoid external locking

Exported types should not expose locking primitives. Relying on the caller to understand the locking protocol and state management of the struct breaks all encapsulation and is much harder to reason about as the scope has increased from the package to all consumers.

Similarly, avoid embedding `sync.Mutex` and friends. When a type is embedded in a struct, the embedded methods are automatically promoted to the hosting struct. That means that ‘Lock’ and ‘Unlock’ become part of the API.

Instead of embedding, declare the mutex in an unexported field.

Consider the consumer

Some of the code we write will be be used by applications and tools. Be mindful and empathetic to their concerns. It's unlikely that applications will be structured the same way we are.
Logging in client packages is a good example. Clients should be free to choose their own logging packages and not be forced to use ours.

If we depend on a logger, we need to define a simple consumer-side contract (ideally compatible with the log package), and allow the user to provide a logger instance to us.

Tests should focus on behavior

The goal of testing is to ensure things function as intended. The goal is not 100% coverage. Simply calling a function will count as coverage but it pointless if it doesn't make assertions about behavior.

- Assertions are required on results
- Interactions with dependencies require assertions

Mocks and fakes can be useful to validate that a test subject is interacting correctly with dependencies. They are just another tool. This does not mean that tests should use mocks exclusively. If a production dependency is lightweight and easy to construct, properly configured production instances should be used.

There is generally a correlation between good test coverage and simple designs. Low coverage is usually more a symptom of poor design than broken code.

Generate mocks and fakes for non-trivial interfaces

While it may be fun to write bespoke, hand-crafted, artisanal mocks, it's might be better to use generated fakes that provide uniform behavior.

We currently use `counterfeiter` and `mockery` to generate fakes and mocks from interfaces. While the generated code provides consistency, the experience of using the tools is less than optimal.

The code generated by `counterfeiter` imports the package where the interface was defined to generate a compile time type assertion. This can result in import cycles if tests are in the same package as the interface. An ugly workaround is to generate the fake from an unexported wrapper interface in an `_test.go` file.

The `mockery` tool does not support mock generation from interfaces defined in test files or from unexported interfaces.

In either case, the tools don't support tests that are in the same package as the interface being mocked.

Avoid path traversal for fixtures and code generation

With the exception of packages imported into source, test fixtures should live in a `testdata` directory with the tests. This helps make relocating the package easier and prevents unexpected coupling to fixtures for other tests.

When generating code, the commands used on the `go:generate` statements should not perform tree traversal. This helps prevent breakage when moving code around.